PYTHON TRAINING

🐍 Introduction to Python Programming

📜 1. History of Python

Python is a high-level, interpreted programming language developed by Guido van Rossum in the late 1980s and officially released in 1991. Van Rossum aimed to create a language that was both powerful and easy to read. The name "Python" was inspired not by the snake, but by the British comedy group Monty Python, reflecting the creator's vision of making programming fun and approachable.

Since its release, Python has undergone multiple major updates, the most significant being the transition from Python 2.x to Python 3.x in 2008, which introduced improved syntax and support for modern programming paradigms.

🌟 2. Key Features

Python has gained global popularity due to its rich set of features:

- **Simple and Readable Syntax**: Resembles natural language, making it beginner-friendly.
- **Interpreted Language**: Code is executed line by line, aiding debugging and testing.
- **Dynamically Typed**: No need to declare variable types.
- **Extensive Standard Library**: Offers modules and tools for file I/O, regular expressions, math, and more.
- **Object-Oriented and Functional**: Supports multiple programming styles.
- **Cross-platform**: Runs seamlessly on Windows, macOS, Linux, and more.

- **Large Community**: Thousands of contributors and vast online resources.

💼 3. Real-World Use Cases

Domain	Applications
Web Development	Backend frameworks like Django, Flask, FastAPI
Data Science	Analysis, visualization using Pandas, NumPy, Matplotlib
Machine Learning & AI	Libraries like TensorFlow, Keras, Scikit-learn
Automation	Scripting for file management, web scraping (e.g., using Selenium, BeautifulSoup)
Cybersecurity	Pen-testing tools and network analysis
Game Development	Simple games using Pygame
Embedded Systems	Programming microcontrollers with MicroPython
Robotics	Used with ROS/ROS2 for robotic control and simulation
Finance	Algorithmic trading, data analysis, risk modeling

🌐 4. Python's Ecosystem

One of Python's strengths lies in its vast ecosystem of third-party libraries and tools. With a package manager like **pip**, developers can access thousands of modules for virtually any task. Some popular packages include:

- `NumPy`, `Pandas`, `SciPy` – for scientific computing
- `Matplotlib`, `Seaborn`, `Plotly` – for data visualization
- `OpenCV` – for computer vision
- `Requests`, `Scrapy`, `BeautifulSoup` – for web scraping
- `Tkinter`, `PyQt`, `Kivy` – for GUI applications
- `PyTorch`, `TensorFlow` – for AI and deep learning

📈 5. Python's Impact and Popularity

Python has become one of the most widely used languages globally. According to rankings by **TIOBE**, **Stack Overflow**, and **GitHub**, Python consistently ranks among the top programming languages. Its rise is largely fueled by the explosion of **data science**, **AI**, and **automation**, areas where Python excels.

In both academia and industry, Python is often the first language taught to students, making it a foundational tool for the next generation of developers and researchers.

🧰 Python Compiler and IDE Installation

🖥️ 1. Python Interpreter (Compiler Equivalent)

Unlike traditional compiled languages like C or Java, Python is an **interpreted language**, meaning the code is executed line by line using an interpreter rather than being compiled into machine code beforehand.

- The standard Python interpreter is called **CPython** – written in C and most widely used.
- Other interpreters include:
    - **PyPy** – Faster execution using Just-In-Time (JIT) compilation.
    - **Jython** – Python implemented in Java.
    - **IronPython** – Python for .NET framework.

To check if Python is installed or to run Python scripts:

```
python --version   # or python3 --version
python script.py   # to execute a script
```

🛠️ 2. Installing Python

🔗 Official Installation

💻*For Windows*

Python can be downloaded from the official website:

- https://www.python.org/downloads/

Steps:

1. Download the installer based on your OS (Windows, macOS, Linux).
2. Run the installer and **check the box** that says *"Add Python to PATH"* (important for using Python from the terminal).
3. Click *Install Now* and complete the setup.

🐧 *For Linux:*

Use a package manager:

```
sudo apt install python3    # Ubuntu/Debian
sudo pacman -S python       # Arch
```

💻 3. Installing an IDE (Integrated Development Environment)

Python can be written in any text editor, but using an IDE greatly enhances productivity with features like syntax highlighting, code suggestions, and debugging tools.

Recommended IDEs for Beginners:

IDE/Text Editor	Key Features
IDLE	Comes bundled with Python, lightweight and simple
VS Code	Free, fast, highly customizable with Python extension
PyCharm	Feature-rich IDE from JetBrains (Community & Pro editions)
Jupyter Notebook	Ideal for data science and visualization workflows
Thonny	Tailored for beginners, clean and minimal UI
Spyder	Popular in scientific computing and data analysis

Installing VS Code (Example):

1. Download from https://code.visualstudio.com

2. Open Extensions panel and install the **Python extension** by Microsoft.

3. It will automatically detect Python installed on your system.

Final Check

After installation:

- Open your IDE
- Create a file `hello.py` and write:

  ```
  print("Hello, Python!")
  ```

- Run the script using the IDE's run button or using the terminal:

  ```
  python hello.py
  ```

🌐 Python Virtual Environment

📌 1. What is a Virtual Environment?

A **Python virtual environment** is an isolated workspace that allows developers to:

- Create and manage separate package installations for different projects.
- Avoid version conflicts between dependencies.
- Keep the global Python installation clean.

This is especially important when working on multiple Python projects that might require **different versions** of libraries such as `Django`, `NumPy`, or `Flask`.

🛠️ 2. Why Use Virtual Environments?

Without a virtual environment, all installed packages are stored globally and can cause:

- **Version clashes** between projects.
- Accidental upgrades or deletions of shared dependencies.
- Difficulty in replicating environments across machines or teams.

Using virtual environments helps maintain **project-specific dependencies**, making collaboration and deployment more reliable.

⚙️ 3. Creating and Using a Virtual Environment

Python includes the built-in module `venv` to manage virtual environments (no need to install anything extra).

✅ Steps to Create and Activate a Virtual Environment:

On Windows:

```
python -m venv myenv       # Create virtual environment named 'myenv'
myenv\Scripts\activate     # Activate the virtual environment
```

On macOS/Linux:

```
python3 -m venv myenv      # Create virtual environment named 'myenv'
source myenv/bin/activate  # Activate the virtual environment
```

Once activated, your terminal will reflect the environment name:

```
(myenv) user@machine:~$
```

📦 4. Installing Packages Inside a Virtual Environment

With the environment activated, any `pip` installation is local to that environment:

```
pip install numpy
pip install flask
```

Installed packages are stored within the `myenv/` directory and won't interfere with system-wide packages.

❌ 5. Deactivating the Virtual Environment

To return to the system's default Python environment, simply run: deactivate

📦 Python Package Manager – pip

🔍 1. What is `pip`?

`pip` stands for **"Pip Installs Packages"** and is the **official package manager for Python**. It allows you to:

- Install external Python libraries from the **Python Package Index (PyPI)**
- Manage dependencies for your projects
- Easily upgrade, remove, or list installed packages

Introduced in Python 3.4+, `pip` comes **pre-installed** with Python distributions.

🔧 2. Why Use `pip`?

Python's true power lies in its **extensive ecosystem** of third-party libraries. `pip` provides a convenient command-line interface to:

- Download and install packages from the internet
- Maintain version control for dependencies
- Automatically resolve package dependencies
- Generate reproducible environments via `requirements.txt`

⚙️ 3. Basic `pip` Commands

Here are some commonly used `pip` commands:

Task	Command
Install a package	`pip install package_name`
Install a specific version	`pip install package_name==1.2.3`
Upgrade a package	`pip install --upgrade package_name`
Uninstall a package	`pip uninstall package_name`
List installed packages	`pip list`
Show package details	`pip show package_name`
Save dependencies to file	`pip freeze > requirements.txt`
Install from requirements.txt	`pip install -r requirements.txt`

🌐 4. Installing from PyPI and Other Sources

By default, `pip` pulls packages from **PyPI (pypi.org)**, the largest Python package repository. You can also install packages from:

- **Git repositories**

pip install git+https://github.com/user/repo.git

- Local directories

pip install ./my_package/

- .whl (wheel) or .tar.gz files

pip install some_package.whl

🛠️ 5. Checking `pip` Installation

You can check if `pip` is installed by running:

bash

```
pip --version
```

If not installed, you can install it using:

bash

```
python -m ensurepip --upgrade
```

Or for older systems:

bash

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py

python get-pip.py
```

🔐 6. Best Practices with pip

- Use `pip` **inside a virtual environment** to avoid affecting global installations.
- Always specify versions in `requirements.txt` for consistency across systems.
- Use `pip freeze` to capture the exact environment for deployment or collaboration.

🧾 1. Python Syntax Overview, Indentation, and Comments

🧠 1.1 Overview of Python Syntax

Python syntax refers to the set of rules that define how Python code must be written and structured. Unlike many programming languages that use symbols like `{}` or `;` to define code blocks and statements, Python emphasizes **readability and clarity** through minimalistic syntax.Python is known for its readable and concise syntax, emphasizing simplicity and clarity. Its design enforces structured code, making it beginner-friendly while powerful for complex applications.

Key features of Python syntax:

- **Statements**: Instructions executed by the interpreter (e.g., assignments, function calls).
- **Variables**: Names bound to data; no explicit type declaration needed (dynamic typing).
- **Blocks**: Groups of statements, defined by indentation rather than braces or keywords.
- **Case Sensitivity**: Python is case-sensitive (Variable ≠ variable).
- **Line Structure**: Statements typically occupy one line, but can be continued using \ or wrapped in parentheses.
- **Keywords**: Reserved words like if, for, while, def, return, etc., control program flow.

- **No Semicolons**: Python doesn't require semicolons to end lines, though they can separate multiple statements on one line.
- English-like keywords (e.g., `if`, `for`, `while`, `def`)
- Code blocks are defined by **indentation** instead of braces

Example:

python

```
def greet(name):
    print("Hello, " + name)
```

⬜ 1.2 Indentation in Python

Indentation in Python is not just for readability — it is **syntactically required**. Indentation is a core feature of Python, defining the scope and hierarchy of code blocks. Unlike languages that use braces {} or keywords like begin/end, Python uses whitespace (spaces or tabs) to group statements.

Rules of Indentation:

- **Consistency**: Use the same number of spaces (typically 4) or tabs for each level. Mixing spaces and tabs is discouraged (Python 3 raises an error for inconsistent mixing).
- **Block Definition**: Code at the same indentation level belongs to the same block.
- **Colon (:)**: A colon after a statement (e.g., if, for, def) indicates a new block, and the next line must be indented.
- **De-indentation**: Returning to a previous indentation level ends the block.
- Mixing tabs and spaces should be avoided.

- Improper indentation will lead to an `IndentationError`.

✅ **Example:**

```
# Correct indentation
if True:
    print("This is indented correctly.")
# Incorrect indentation
if True:
print("This will raise an IndentationError.")
```

Conditional Statement:

```
x = 10
if x > 5:
   print("x is greater than 5")  # Indented: part of if block
   print("This is still in the block")
print("This is outside the if block")  # De-indented: outside block
```

Program Flow: If x > 5, the indented block executes; then flow continues to the de-indented statement.

Data Flow: x is checked, and strings are printed based on the condition.

Loop:

```
for i in range(3):
    print(f"Number: {i}")  # Indented: loop body
    if i == 1:
        print("i is 1")    # Further indented: nested block
print("Loop ended")       # De-indented: after loop
```

Program Flow: The for loop runs three times, executing the indented block; nested if runs when i == 1.

Data Flow: i iterates over [0, 1, 2], and strings are printed.

Error Example (Incorrect Indentation)

```
if True:
    print("Correct")
     print("This will cause an IndentationError")  # Inconsistent indent
```

🧩 **Use in Control Structures:**

python

```
x = 10
if x > 5:
    print("x is greater than 5")
    print("This is part of the same block")
print("This is outside the if block")
```

Best Practices

- Use **4 spaces** per indentation level (PEP 8 standard).
- Configure editors to convert tabs to spaces.

- Avoid excessive nesting to keep code readable.
- Be consistent within a file to prevent errors.

💬 1.3 Comments in Python

Comments are used to make the code more understandable by explaining what specific blocks of code do. Python ignores comments during execution.

There are two types of comments:

🔸 **1. Single-line Comments**

Begin with the hash (#) symbol. Anything after # is ignored by the interpreter. Used for brief explanations or notes.

python

```
# This is a single-line comment

name = "Alice"  # This is an inline comment
```

🔸 **2. Multi-line (Block) Comments**

Python does not have a built-in syntax for multi-line comments like `/* */`. However, you can use a series of # symbols or **multi-line strings** (though the latter is technically not a comment but a string that is not assigned or used).

python

```
# This is a multi-line comment

# written across several lines

"""
```

```
This is a multi-line string

which can act like a comment

if not assigned to any variable.

"""
```

🧪 Best Practices

- Always use **meaningful comments** to explain complex logic or assumptions.
- Maintain consistent **indentation** (typically 4 spaces per level).
- Avoid unnecessary comments for obvious code.
- Use comments to **improve code readability**, especially in collaborative environments.
- **Clarity**: Write concise, meaningful comments explaining *why* code exists, not just *what* it does.
- **Avoid Redundancy**: Don't comment obvious code (e.g., # Set x to 5 for x = 5).
- **Use for Documentation**: Comment complex logic, functions, or tricky sections.
- **Triple Quotes for Docstrings**: Use """ at the start of functions or modules for documentation, not as generic comments.

🖨️ 2. Input/Output (I/O) Operations in Python

📥 2.1 Introduction to I/O in Python

Input/Output (I/O) operations are essential for interacting with the user and the outside world. Python provides simple, readable methods to **accept input from the user** and **display output** to the screen or other devices. Additionally, Python allows reading from and writing to **files**, which is also part of I/O operations.These operations are essential for dynamic programs, enabling data exchange and persistence. Python provides simple, built-in functions and modules for I/O, tightly integrated with program flow constructs.

📌 2.2 Output Operations (`print()`)

The `print()` function is used to display output to the standard output device, usually the **screen**.

✅ **Syntax:**

python

```
print(object(s), sep=' ', end='\n', file=sys.stdout)
```

🔹 **Examples:**

python

```
print("Hello, World!")                    # Basic output

print("Python", "is", "fun")              # Multiple values
```

```
print("Hello", end="!")                          # Custom end character

print("Value:", 42, sep=" --> ")                 # Custom separator
```

🔹 **Output Formatting:**

Python supports **string formatting** using:

- **f-strings (Python 3.6+)**
- `format()` method
- Old-style `%` formatting

python

```
name = "Alice"

age = 25

print(f"My name is {name} and I am {age} years old.")     # f-string

print("My name is {} and I am {} years old.".format(name, age))  #
format()
```

📥 2.3 Input Operations (`input()`)

The `input()` function is used to take input from the user as a **string**.

input(prompt): Reads a line from the user as a string.

- prompt: Optional string displayed to the user.

- Always returns a string, requiring type conversion (e.g., int() or float()) for non-string data.

✅ **Syntax:**

python

```
variable = input("Prompt message: ")
```

🔹 **Example:**

python

```
name = input("Enter your name: ")

print("Hello, " + name + "!")
```

⚠️ **Note**

All input from `input()` is read as a **string**. For numerical operations, you need to **typecast**:

python

```
age = int(input("Enter your age: "))      # Converts string to integer

height = float(input("Enter your height: "))  # Converts string to float
```

Error Handling: Since input() returns a string, converting to numbers requires error handling.

```
try:
    age = int(input("Enter your age: "))  # Convert string to int
    print(f"You are {age} years old.")
except ValueError:
    print("Please enter a valid number.")
```

📁 2.4 File I/O Operations

Python also provides powerful tools to handle **file reading and writing**. File I/O involves reading from and writing to files on the filesystem, enabling data storage and retrieval.

🔹 **Opening a File:**

open(filename, mode, encoding=None): Opens a file and returns a file object.

- filename: Path to the file (e.g., "data.txt").
- mode: Specifies operation:
    - "r": Read (default).
    - "w": Write (overwrites file).
    - "a": Append (adds to end).
    - "r+": Read and write.
    - Add "b" for binary mode (e.g., "rb" for binary read).
- encoding: Text encoding (e.g., "utf-8" for text files).

python

```
file = open("filename.txt", "mode")
```

```
file = open("example.txt", "w", encoding="utf-8")
```

```
file.write("Hello, file!\n")
```

```
file.close()
```

Modes:

Mode	Description
`'r'`	Read (default)
`'w'`	Write (overwrite)
`'a'`	Append
`'b'`	Binary mode
`'+'`	Read and write combined

- **Writing to a File:**

write(text): Writes a string to the file.

writelines(lines): Writes a list of strings without adding newlines.

python

```
file = open("example.txt", "w")

file.write("Hello File!\n")

file.close()
```

- **Reading from a File:**

read(): Reads the entire file as a string.

readline(): Reads one line (up to newline).

readlines(): Reads all lines into a list.

Iterable: Loop over the file object to read line by line (memory-efficient).

python

```
file = open("example.txt", "r")
content = file.read()
print(content)
file.close()
```

- **Error Handling:**

Handle missing files or I/O issues.

```
try:
    with open("nonexistent.txt", "r", encoding="utf-8") as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("File not found.")
except IOError:
```

```
    print("An error occurred while reading the file.")
```

🔹 **Best Practice: Using `with` Statement**

The `with` statement automatically closes the file after use:

python

```
with open("example.txt", "r") as file:
    data = file.read()
    print(data)
```

🧪 Combining I/O with Program Flow

```
# Program to store numbers from 1 to n (user input) in a file, skipping multiples of 3
def write_non_multiples_of_three(n, filename):
    """Write numbers from 1 to n to a file, skipping multiples of 3."""
    with open(filename, "w", encoding="utf-8") as file:
        for i in range(1, n + 1):
            if i % 3 == 0:  # Skip multiples of 3
                continue
            file.write(f"{i}\n")  # Write number to file

# Read and print numbers from file
def read_numbers(filename):
    """Read numbers from file and print them."""
    try:
```

```
        with open(filename, "r", encoding="utf-8") as file:
            print("Numbers from file:")
            for line in file:
                print(line.strip(), end=", ")
    except FileNotFoundError:
        print("File not found.")
    except IOError:
        print("Error reading file.")

# Main program with user input
try:
    n = int(input("Enter a positive integer: "))  # Get user input
    if n <= 0:
        print("Please enter a positive integer.")
    else:
        filename = "non_multiples.txt"
        write_non_multiples_of_three(n, filename)  # Write to file
        read_numbers(filename)                     # Read and print
except ValueError:
    print("Invalid input. Please enter a valid integer.")
```

✅ Conclusion

Python simplifies I/O operations with intuitive functions like `print()` and `input()`, and offers robust file-handling capabilities. These features make it easy to build interactive programs and handle data storage efficiently.

➕ 3. Operators in Python

🧠 What Are Operators?

Operators are special symbols or keywords in Python used to perform operations on variables and values. They are the building blocks of expressions and allow us to manipulate data logically, mathematically, or relationally. They are fundamental to manipulating data and controlling program flow in Python, often used within conditionals, loops, and expressions. Python's operators are categorized based on their functionality.

🧮 3.1 Arithmetic Operators

Used to perform **basic mathematical operations**.

Operator	Description	Example	Result
`+`	Addition	`5 + 3`	`8`
`-`	Subtraction	`5 - 2`	`3`
`*`	Multiplication	`4 * 2`	`8`
`/`	Division (float)	`5 / 2`	`2.5`
`//`	Floor division	`5 // 2`	`2`
`%`	Modulus (remainder)	`5 % 2`	`1`
`**`	Exponentiation	`2 ** 3`	`8`

```
# Calculate sum and average in a loop

total = 0

for i in range(1, 4):  # 1, 2, 3

    total += i        # += is shorthand for total = total + i

average = total / 3

print(f"Sum: {total}, Average: {average}")  # Output: Sum: 6, Average: 2.0
```

🧾 3.2 Assignment Operators

Used to **assign values** to variables and update them.

Operator	Description	Example	Equivalent To
`=`	Assign value	`x = 5`	–
`+=`	Add and assign	`x += 3`	`x = x + 3`
`-=`	Subtract and assign	`x -= 2`	`x = x - 2`
`*=`	Multiply and assign	`x *= 4`	`x = x * 4`
`/=`	Divide and assign	`x /= 2`	`x = x / 2`
`//=`	Floor divide & assign	`x //= 2`	`x = x // 2`
`%=`	Modulus and assign	`x %= 2`	`x = x % 2`

`**=`	Power and assign	`x **= 3`	`x = x ** 3`

```
# Accumulate sum
total = 0
for i in range(1, 6):
    total += i  # Same as total = total + i
print(total)    # Output: 15
```

🧪 3.3 Comparison (Relational) Operators

Used to **compare values**, returning True or False, often in conditional expressions.

Operator	Description	Example	Result
`==`	Equal to	`5 == 5`	`True`
`!=`	Not equal to	`5 != 3`	`True`
`>`	Greater than	`5 > 3`	`True`
`<`	Less than	`3 < 5`	`True`
`>=`	Greater or equal	`5 >= 5`	`True`
`<=`	Less or equal	`4 <= 5`	`True`

```
# Check if number is positive
num = int(input("Enter a number: "))
if num > 0:
    print("Positive")
elif num == 0:
    print("Zero")
else:
    print("Negative")
```

🔁 3.4 Logical Operators

Combine boolean expressions, used in complex conditions.

Operator	Description	Example	Result
`and`	True if both are True	`(x > 5 and x < 10)`	`True`
`or`	True if at least one is True	`(x < 5 or x > 15)`	`True`
`not`	Inverts the condition	`not(x > 5)`	`False`

```
# Check if number is in range
num = 15
if num >= 10 and num <= 20:
    print("In range")
else:
```

```
    print("Out of range")
```

🧱 3.5 Bitwise Operators

Operate at the **binary level** (useful in low-level programming).

Operator	Description	Example	Binary Operation
`&`	AND	`5 & 3`	`0101 & 0011 = 0001` (1)
`	`	`	OR	`5
`^`	XOR	`5 ^ 3`	`0101 ^ 0011 = 0110` (6)
`~`	NOT (invert bits)	`~5`	`~0101 = 1010` (-6)
`<<`	Left shift	`5 << 1`	`1010` (10)
`>>`	Right shift	`5 >> 1`	`0010` (2)

```
# Check if number is odd using bitwise AND
num = 7
if num & 1:  # 1 if odd, 0 if even
    print("Odd")
else:
    print("Even")
```

🔎 3.6 Identity Operators

Used to **compare object identities**, not just values; ie Check if two variables refer to the same object in memory.

Operator	Description	Example	Result
`is`	True if same object	`a is b`	`True`
`is not`	True if different object	`a is not b`	`True`

Example

```
a = [1, 2, 3]
b = a
c = [1, 2, 3]
print(a is b)   # True (same reference)
print(a is c)   # False (different objects, same content)
```

🧰 3.7 Membership Operators

Check for **membership** in a sequence (like lists, strings, tuples).

Operator	Description	Example	Result
`in`	True if value is present	`'a' in 'apple'`	`True`
`not in`	True if not present	`10 not in [1,2,3]`	`True`

```
# Check vowel in string
char = input("Enter a character: ")
if char.lower() in "aeiou":
    print("Vowel")
else:
    print("Not a vowel")
```

📌 3.8 Operator Precedence

Operators have a hierarchy determining evaluation order (highest to lowest priority):

1. ** (exponentiation)
2. ~, +, - (unary operators)
3. *, /, //, %
4. +, - (binary addition/subtraction)
5. <<, >>
6. &
7. ^
8. |
9. Comparison (==, !=, >, <, >=, <=)
10. is, is not, in, not in
11. not
12. and
13. or

- Use parentheses () to override precedence for clarity.

```
result = 5 + 3 * 2  # * before +, result = 11

result2 = (5 + 3) * 2  # Parentheses first, result = 16

print(result, result2)  # Output: 11 16
```

✅ Conclusion

Python provides a comprehensive set of operators to perform operations across data types — from numbers and strings to complex objects. Understanding these operators is fundamental to writing expressive, efficient, and logical Python code.

🗂️ 4. Data Types and Data Structures in Python

🧠 Introduction

In Python, **data types** define the kind of values a variable can hold and the operations supported, while **data structures** are ways to organize and store data for efficient access and manipulation. Python's dynamic typing means types are inferred at runtime without explicit declarations. Data types are broadly classified into **primitive** (basic, indivisible) and **non-primitive** (complex, composed of other types).

Python's data types are broadly categorized into:

- **Primitive Data Types** – Basic building blocks
- **Non-Primitive Data Structures** – Used to store multiple values and complex relationships

---

A. Primitive Data Types

These are the most basic types of data that represent **single values**. Primitive data types are the fundamental building blocks, representing single, indivisible values. They are simple, with fixed memory sizes and direct operations.

4.1 Integer (`int`)

Description: Represents whole numbers, positive or negative, with no size limit in Python.

Characteristics:

- Immutable: Value cannot change once set (a new object is created for changes).
- Supports arithmetic operations (+, -, *, /, //, %, **).

Usage: Stores counts, indices, or exact quantities.

Example

```
age = 25
```

```
count = 42
```

```
negative = -100
```

4.2 Float (`float`)

Description: Represents decimal numbers (floating-point).

Characteristics:

- Immutable.
- Supports arithmetic operations; may have precision limits for very large/small values.
- Can use scientific notation (e.g., 2.5e3 for 2500.0).

Usage: Stores measurements, percentages, or approximate values.

Example

```
price = 99.99
```

```
pi = 3.14159
```

```
temperature = -10.5
```

4.3 String (`str`)

Description: An immutable sequence of Unicode characters.

Characteristics:

- Enclosed in quotes (', ", ''', """).
- Supports indexing (s[0]), slicing (s[1:3]), and operations like concatenation (+) or repetition (*).
- Immutable: Modifications create new strings.

Usage: Stores text, such as names or messages.

Example

```
name = "Alice"
```

4.4 Boolean (`bool`)

Description: Represents logical values: True or False.

Characteristics:

- Immutable.
- Subtype of int (True is 1, False is 0).
- Result of comparison (==, >) or logical operations (and, or, not).
- Falsy values include 0, 0.0, "", [], {}, None; others are truthy.

Usage: Flags or logical conditions.

Example

```
is_active = True

has_error = False
```

4.5 NoneType (`None`)

Represents a null or no-value condition.

Example

```
value = None
```

🔸 B. Non-Primitive Data Structures

Non-primitive data types are complex, composed of primitive types or other non-primitive types. They are designed to store and organize multiple values, supporting advanced operations like indexing, slicing, or key-based access. These are also referred to as data structures due to their role in data organization.

4.6 List (`list`)

Description: A mutable, ordered sequence of elements (any type).

Characteristics:

- Enclosed in square brackets [].
- Supports indexing, slicing, and modification (e.g., append(), pop(), insert()).
- Elements can be heterogeneous (e.g., [1, "a", 3.14]).

Usage: Stores collections like scores, items, or mixed data.

- Can contain mixed data types

- Supports indexing and slicing

```
numbers = [1, 2, 3, 4]

mixed = [42, "Python", 3.14]

fruits = ["apple", "banana", "cherry"]

fruits[0] = "mango"  # Lists are mutable
```

4.7 Tuple (`tuple`)

Description: An immutable, ordered sequence of elements (any type).

Characteristics:

- Enclosed in parentheses () or created without (e.g., 1, 2).
- Supports indexing and slicing but no modification.
- Often used for fixed data or unpacking (e.g., x, y = point).

Usage: Stores constant collections or grouped values.

Example

- Useful for fixed data
- Can contain mixed types
- Faster than lists due to immutability

```
# Fixed data

coordinates = (3, 4)

rgb = (255, 128, 0)
```

4.8 Set (`set`)

Description: A mutable, unordered collection of unique, immutable elements.

Characteristics:

- Enclosed in curly braces {} or created with set().
- Supports operations like union (|), intersection (&), difference (-).
- No indexing or slicing due to unordered nature.
- Elements must be hashable (e.g., int, str, tuple).

Usage: Stores unique items, like IDs or tags.

- No duplicate items allowed
- No indexing or slicing
- Useful for membership tests and mathematical set operations

Example

```
unique_ids = {101, 102, 103}

# Unique elements

tags = {"red", "blue", "green"}

empty_set = set()
```

4.9 Dictionary (`dict`)

- **Description**: A mutable collection of key-value pairs.
- **Characteristics**:
  - Enclosed in curly braces {} (e.g., {"key": value}).
  - Keys must be immutable (e.g., str, int, tuple); values can be any type.
  - Supports key-based access (d["key"]), addition, and removal.
  - Unordered (Python 3.7+ preserves insertion order but not guaranteed for logic).
- **Usage**: Stores mappings like settings, records, or lookup tables.
- Keys must be unique and immutable
- Values can be any data type
- Dynamic and highly optimized for lookups

Examples

```
student = {"name": "Alice", "age": 22}

print(student["name"])  # Access via key
```

4.10Frozen Set (frozenset)

- **Description**: An immutable version of a set.
- **Characteristics**:
  - Created with frozenset() (e.g., frozenset([1, 2, 3])).
  - Supports same operations as set but cannot be modified.
  - Can be used as dictionary keys or set elements.
- **Usage**: Stores constant unique collections.

```
# Immutable unique elements

fixed_nums = frozenset([1, 2, 3])
```

C. Key Characteristics and Differences

- Primitive:
    - Simple, single values (int, float, complex, bool).
    - Immutable, lightweight, and used for basic computations.
    - Directly manipulated with operators (e.g., +, ==).
- Non-Primitive:
    - Complex, store multiple values (str, list, tuple, dict, set, frozenset, NoneType).
    - Mutable (list, dict, set) or immutable (str, tuple, frozenset, None).
    - Support specialized operations (e.g., indexing for sequences, key access for dict, union for set).
- Data Structures:
    - Non-primitive types like list, tuple, dict, and set are data structures because they organize data for efficient access.
    - str is a sequence structure; dict is a mapping structure; set is a collection structure.

D. Type Checking and Conversion

- Check Type:
    - type() returns the type (e.g., type(42) → <class 'int'>).
    - isinstance(value, type) checks if a value matches a type (e.g., isinstance("hello", str) → True).
- Convert Types:
    - Explicit conversion with int(), float(), str(), list(), tuple(), set(), dict(), etc.

```
# Conversions

num = int(3.14)       # float to int: 3

text = str(42)        # int to str: "42"
```

```
lst = list("abc")      # str to list: ["a", "b", "c"]

tup = tuple([1, 2])    # list to tuple: (1, 2)

st = set([1, 1, 2])    # list to set: {1, 2}
```

E. Memory and Mutability

- **Immutable Types** (int, float, complex, bool, str, tuple, frozenset, None):
    - Cannot be modified in-place; operations create new objects.
    - Example: Concatenating strings ("a" + "b") creates a new string "ab".
- **Mutable Types** (list, dict, set):
    - Can be modified in-place (e.g., lst.append(1) changes lst).
    - Example: Adding to a dictionary (d["key"] = value) updates d.
- **Memory**:
    - Primitive types use fixed memory (e.g., int size depends on value but managed internally).
    - Non-primitive types grow dynamically (e.g., list expands with append()).

F. Example Assignments (No I/O or Flow)

To illustrate without conditionals, loops, or I/O, here are assignments showcasing data types:

```
# Primitive types

counter = 100          # int

price = 19.99          # float

vector = 1 + 2j        # complex

is_valid = False       # bool
```

```
# Non-primitive types
message = "Welcome"     # str
items = [10, 20, 30]    # list
point = (5, 6)          # tuple
info = {"id": 1, "name": "Bob"}  # dict
colors = {"red", "blue"}         # set
fixed_ids = frozenset([7, 8])    # frozenset
placeholder = None               # NoneType

# Conversions
num_str = str(counter)         # int to str: "100"
float_num = float(price)       # float: 19.99
list_chars = list(message)     # str to list: ["W", "e", "l", "c", "o", "m", "e"]
tuple_items = tuple(items)     # list to tuple: (10, 20, 30)
set_nums = set(items)          # list to set: {10, 20, 30}
```

🧮 Type Conversion

Python allows you to **convert** between types using built-in functions:

Example

```
int("10")        # Converts string to integer

str(99.9)        # Converts float to string

list("abc")      # Converts string to list: ['a', 'b', 'c']
```

📋 Summary Table

Category	Type	Description	Mutable
Primitive	`int`	Whole numbers	❌
	`float`	Decimal numbers	❌
	`str`	Text strings	❌
	`bool`	Boolean values	❌
	`NoneType`	Represents null/undefined	❌
Non-Primitive	`list`	Ordered, mutable sequence	✅

`tuple`	Ordered, immutable sequence	❌
`set`	Unordered, unique elements	✅
`dict`	Key-value mappings	✅

✅ Conclusion

Understanding Python's **data types and data structures** is essential for effective programming. Whether you're handling a single value or a complex dataset, Python provides the right structure to store, access, and manipulate data efficiently.

🧾 Basic Python Program Demonstrating Data Types, Variables, Operators, and I/O

```
# ---------------------------------------------
# Python Program to Demonstrate:
# - Variable declaration
# - Primitive & non-primitive data types
# - Input/output operations
# - Operators (arithmetic, logical, comparison)
# - Comments and indentation
# ---------------------------------------------

# 🔸 Output operation using print()
print("Welcome to Python Basics Demonstration!\n")

# 🔹 Variable declaration with primitive data types
name = input("Enter your name: ")           # String (str)
age = int(input("Enter your age: "))        # Integer (int)
height = float(input("Enter your height: "))  # Float (float)
is_student = True                           # Boolean (bool)

# 🔸 Displaying variable types (output and type checking)
print(f"\nHello {name}, here are your details:")
print("Age:", age, "| Type:", type(age))
print("Height:", height, "| Type:", type(height))
print("Student:", is_student, "| Type:", type(is_student))
```

```
# 🔹 Arithmetic operations
year_of_birth = 2025 - age                  # Using subtraction operator
bmi = height / (1.75 ** 2)                  # Division and exponentiation
print(f"Estimated Year of Birth: {year_of_birth}")
print(f"Estimated BMI (with dummy height 1.75m): {round(bmi, 2)}")

# 🔹 Comparison and logical operators
is_adult = age >= 18                        # Comparison operator
eligible = is_adult and is_student          # Logical AND operator
print(f"Are you an adult student? {eligible}")

# 🔹 Assignment operators
score = 10
score += 5                                  # Equivalent to score = score + 5
print("Score after bonus points:", score)

# 🔹 Using non-primitive data structures

# List: ordered and mutable
subjects = ["Math", "Science", "Python"]
subjects.append("AI")                       # Adding new subject
print("\nSubjects List:", subjects)

# Tuple: ordered and immutable
coordinates = (10.5, 20.5)
print("Coordinates (Tuple):", coordinates)
```

```
# Set: unordered and unique
skills = {"Python", "ML", "Python"}          # Duplicate "Python" will be removed
print("Unique Skills (Set):", skills)

# Dictionary: key-value pairs
student_info = {
    "Name": name,
    "Age": age,
    "Is_Student": is_student,
    "Subjects": subjects
}
print("Student Info (Dictionary):", student_info)

# 🔹 Type conversion example
age_str = str(age)                          # Convert int to string
print("Age as string:", age_str, "| Type:", type(age_str))

# 🔹 NoneType
undefined_value = None
print("Undefined variable (NoneType):", undefined_value, "| Type:", type(undefined_value))

# ✅ End of program
print("\nThank you for using the demo program!")
```

🧠 5. Program Flow and Data Flow in Python

Program flow refers to the order in which a program's instructions are executed. Program flow refers to the **order in which individual statements, instructions, or function calls are executed in a program**. In Python, you control the flow of your program using **conditional statements**, **loops**, and **control statements**. These tools allow you to decide which code to execute based on certain conditions, repeat actions, and exit loops early if necessary.Python uses a sequential flow by default, but you can alter it using:

- **Conditional statements** (if, elif, else)
- **Loops** (for, while)
- **Control statements** (break, continue, pass, return)

Data Flow

Describes how data is created, transformed, and passed through a program, from inputs to outputs, as variables are manipulated by statements. It's influenced by:

- Variables and assignments
- Function calls and return values
- Control structures that direct data processing

Conditional statements, **looping**, and **control statements** are key mechanisms for controlling both flows, enabling decision-making, repetition, and interruption of execution.

5.1 Conditional Statements

Conditional statements allow decisions based on conditions. Conditional statements execute specific blocks of code based on whether a condition evaluates to True or False. They direct program flow by branching and transform data by selectively processing it.

Syntax:

- if: Evaluates a condition; executes its block if True.
- elif: Checks additional conditions if previous ones are False.
- else: Executes if no prior conditions are True (optional).
- Indentation (4 spaces) defines the block scope.

Program Flow:

- The interpreter evaluates conditions in order (if, then elif, then else).
- Only the first True condition's block executes; others are skipped.
- Execution resumes after the conditional block.

Data Flow:

- Input data is tested (e.g., via comparison operators like >, ==).
- Variables may be modified or new values computed in the executed block.
- Output depends on which block runs.

📝 **Syntax:**

```
if condition:
    # Executes if condition is True
elif another_condition:
    # Executes if the previous condition is False and this one is True
else:
    # Executes if all conditions are False
```

- **Example:**

if, elif, else:

```
# Determine grade category

score = 85

category = None

if score >= 90:

    category = "Excellent"

elif score >= 80:

    category = "Good"

elif score >= 70:

    category = "Fair"

else:

    category = "Needs Improvement"
```

Data flow: age is checked against conditions, and only one print statement executes.

Program flow: Execution skips to after the block once a condition is met or if no conditions are true.

`if`: Checks if the condition is **True**.

`elif`: Else if the previous condition was **False** and the new condition is **True**.

`else`: Executes when all previous conditions are **False**.

5.2 Nested conditionals:

Example

```
# Check number properties

number = 10

result = ""

if number > 0:

    result = "Positive"

    if number % 2 == 0:

        result = "Positive Even"

else:

    result = "Non-positive"
```

Program Flow: number > 0 (True) → inner number % 2 == 0 (True) → sets result; skips else.

Data Flow: number (10) → tested → result set to "Positive Even".

5.3 Looping Statements (for, while)

Loops repeat a block of code multiple times, either for a fixed number of iterations or until a condition is met. They control program flow by cycling and manage data flow by processing sequences or updating variables.

- **Types**:
    - **for Loop**:
        - Iterates over a sequence (e.g., list, range, string).
        - Syntax: for variable in sequence:.
    - **while Loop**:
        - Repeats as long as a condition is True.
        - Syntax: while condition:.
- **Program Flow**:
    - **for**: Executes the block for each item in the sequence, then exits.
    - **while**: Checks the condition before each iteration; exits when False.
    - Indentation defines the loop body.
    - Loops can be nested for complex iterations.
- **Data Flow**:
    - **for**: Each iteration processes an element, updating variables or accumulating results.
    - **while**: Variables are modified until the condition changes, often based on computed values.
    - Data is transformed iteratively (e.g., summing, filtering).

```
# Sum a sequence

numbers = [1, 2, 3, 4]

total = 0

for num in numbers:

    total = total + num
```

Program Flow: Iterates over [1, 2, 3, 4], executing the block 4 times, then exits.

Data Flow: numbers → num takes each value → total accumulates (1, 3, 6, 10).

```
# Double until limit

value = 1

limit = 10

while value < limit:

    value = value * 2
```

Program Flow: Checks value < limit, executes block (doubles value), repeats until False (stops at 16).

Data Flow: value (1) → doubled (2, 4, 8, 16) → stops when value >= limit.

```
# Create pairs

pairs = []

for i in [1, 2]:

    for j in [3, 4]:

        pairs = pairs + [(i, j)]
```

Program Flow: Outer loop runs for i=1, 2; inner loop runs for j=3, 4 each time; total 4 iterations.

Data Flow: i, j → combined as tuples → appended to pairs ([(1, 3), (1, 4), (2, 3), (2, 4)]).

5.4 Control Statements (break, continue, pass)

Control statements modify the execution of loops or functions, interrupting or altering program flow and affecting how data is processed.

- **Types**:
  - **break**: Exits the innermost loop immediately.
  - **continue**: Skips the rest of the current loop iteration, moving to the next.
  - **pass**: Does nothing; a placeholder for empty blocks.
  - **return**: Exits a function, optionally returning a value.
- **Program Flow**:
  - **break**: Terminates the loop, resuming after it.
  - **continue**: Jumps to the loop's next iteration.
  - **pass**: Allows execution to proceed without action (used to avoid syntax errors).
  - **return**: Stops function execution, passing control back to the caller.
- **Data Flow**:
  - **break**: Halts data processing early.
  - **continue**: Skips processing for specific iterations, affecting output.
  - **pass**: No data impact; maintains structure.
  - **return**: Transfers computed data to the caller.

```
# Stop at first negative
values = [5, 3, -1, 7]
first_negative = None
for val in values:
    if val < 0:
        first_negative = val
        break
```

Program Flow: Iterates until val = -1, then break exits loop.

Data Flow: values → val checked → first_negative set to -1.

```
# Collect non-zero numbers
numbers = [0, 2, 0, 4]
non_zeros = []
for num in numbers:
    if num == 0:
        continue
    non_zeros = non_zeros + [num]
```

Program Flow: Skips num == 0 iterations; appends others.

Data Flow: numbers → num → non_zeros gets [2, 4].

```
# Placeholder for future logic
status = "pending"
if status == "pending":
    pass  # To be implemented later
else:
    status = "done"
```

Program Flow: pass does nothing; skips block for "pending".

Data Flow: status unchanged ("pending").

```
# Define a function
def get_first_even(numbers):
    result = 0
```

```
    for num in numbers:
        if num % 2 == 0:
            result = num
            return result
    return result
numbers = [1, 3, 4, 6]
even = get_first_even(numbers)
```

Program Flow: Loops until first even number (4), return exits function.

Data Flow: numbers → num → result set to 4 → assigned to even.

5.4 Key Interactions

- **Conditionals in Loops**:
    - Combine to filter or process data selectively.
    - Example: if inside for to skip or modify based on conditions.
- **Control Statements in Loops**:
    - break and continue refine iteration, reducing unnecessary work.
    - Example: break to stop early, continue to skip invalid data.
- **Functions with return**:
    - Encapsulate flow, using return to pass processed data.
    - Integrate conditionals and loops for complex logic.
- **Indentation**:
    - 4 spaces per level (e.g., loop body, nested if).
    - Ensures clear block boundaries, critical for flow.
- **Comments**:
    - Explain logic (e.g., # Stop at first negative).
    - Improve readability without affecting execution.

5.5 Python Program Demonstrating Program Flow

```
# ---------------------------------------------
# Python Program Demonstrating:
# - Conditional Statements (if, elif, else)
# - Loops (for, while)
# - Control Statements (break, continue, pass)
# ---------------------------------------------

# 🔸 Input and initialization
age = int(input("Enter your age: "))  # Age input for conditional checks
count = 0  # Counter variable for loop demonstration

# 🔹 Conditional Statements (if-elif-else)
if age >= 18:
    print("You are an adult!")
elif age >= 13:
    print("You are a teenager.")
else:
    print("You are a child.")

# 🔹 For loop (iterating over a sequence)
print("\nFruits you like:")
fruits = ["apple", "banana", "cherry", "orange", "grape"]
for fruit in fruits:
    if fruit == "orange":
```

```
        continue  # Skip "orange" and move to the next fruit
    print(fruit)

# 🔹 While loop (counting down)
print("\nCountdown:")
while count < 5:
    print(count)
    count += 1
    if count == 3:
        break  # Stop the loop when count reaches 3

# 🔹 Control statement (pass in function)
def sample_function():
    pass  # This function does nothing but syntactically required

# 🔸 Final output
print("\nProgram has ended successfully!")
```

5.6 Explanation of the Program:

Conditional Statements: Based on the user's age, the program checks which category they belong to (adult, teenager, or child) using if, elif, and else. For Loop: The program iterates through a list of fruits and prints each fruit, except for "orange" (which is skipped using continue). While Loop: A counter is used to print numbers from 0 to 2, then the loop is stopped prematurely when the count reaches 3 using break. Control Statement (pass): A function sample_function() is defined but does nothing due to the pass statement, acting as a placeholder for future functionality.

5.7 Key Points

- **Program flow** is controlled by the structure of conditionals, loops, and control statements.
- **Data flow** depends on how variables are assigned, modified, and passed through these structures.
- Python's indentation enforces block structure, ensuring clear flow.
- Use break and continue sparingly to avoid complex logic.
- pass is useful for planning code without breaking execution.

5.8 Conclusion

The program flow is a critical aspect of Python programming. With conditional statements, loops, and control flow statements, Python allows you to write dynamic, responsive code that reacts to user input and data in a logical manner. Mastering these basic building blocks is essential for building more complex applications.

🧩 6. Functions in Python

6.1 Objective:

In this module, you will gain a clear understanding of how to define and use functions in Python. You will explore:

- Function syntax and arguments
- Scope of variables
- Lambda functions with functional programming tools
- DocStrings for documentation
- Working with modules and standard libraries

6.2 What is a Function?

A **function** is a reusable block of code that performs a specific task. Functions help in making the code modular, easy to understand, and reusable. They control **program flow** by executing when called and manage **data flow** by processing inputs and returning outputs. Functions in Python support various argument types, anonymous definitions (lambda), and integration with modules.

6.3 Syntax of a Function

Functions are defined using the def keyword, followed by a name, parameters (optional), and a body. They can return values using return.

```
def function_name(parameters):
    """Optional docstring describing the function."""
    # Function body (indented, 4 spaces)
    statement1
    statement2
    return value  # Optional; defaults to None if omitted
```

def: Keyword to define a function.

function_name: Identifier (follows variable naming rules, e.g., lowercase_with_underscores).

parameters: Optional inputs (zero or more).

:: Introduces the indented function body.

return: Outputs a value; exits the function.

Example:

```
def greet(name):
    """This function greets the user."""
    return f"Hello, {name}!"
print(greet("Alice"))
```

Example:

```
def add_numbers(a, b):
   """Return the sum of two numbers."""
```

```
    result = a + b
    return result
```

Program Flow: Define function → call it (e.g., add_numbers(2, 3)) → execute body → return value.

Data Flow: Inputs a, b → summed → result → returned.

Calling a Function:

```
sum_value = add_numbers(5, 10)  # Calls function, assigns 15 to sum_value
```

No Parameters:

```
def greet():
    """Return a fixed greeting."""
    return "Hello!"
```

6.4 Functions with *args and **kwargs

Python supports flexible argument handling for functions that accept variable numbers of inputs.

`*args`

- Allows a function to accept any number of **positional arguments** as a tuple.
- Useful for functions needing a variable-length input list.
- Syntax: *args in parameter list (conventionally named args).

```
def sum_all(*args):
    """Sum all positional arguments."""
    total = 0
    for num in args:
```

```
    total += num
  return total
```

Program Flow: Collects arguments into args tuple → iterates → sums → returns.

Data Flow: Inputs (e.g., 1, 2, 3) → args = (1, 2, 3) → total = 6.

```
result = sum_all(1, 2, 3, 4)  # Returns 10
result_empty = sum_all()      # Returns 0
```

`**kwargs`

- Allows a function to accept any number of **keyword arguments** as a dictionary.
- Useful for named parameters with flexible keys.
- Syntax: **kwargs in parameter list (conventionally named kwargs).

🧪 Example:

```
def show_info(*args, **kwargs):
    print("Arguments (args):", args)
    print("Keyword Arguments (kwargs):", kwargs)
show_info("Python", 3.10, language="English", level="Beginner")
```

🧪 Combining Parameters:

- Order: regular parameters, *args, **kwargs.

```
def combine(a, b, *args, **kwargs):
  """Combine all inputs."""
  result = [a, b]
  result.extend(args)
```

```
    result.append(kwargs)
    return result
```

Program Flow: Assign a, b → collect args → collect kwargs → combine → return.

Data Flow: a=1, b=2, 3, 4, x="extra" → [1, 2, 3, 4, {"x": "extra"}].

```
output = combine(1, 2, 3, 4, x="extra")  # Returns [1, 2, 3, 4, {"x": "extra"}]
```

6.5 Scope of Variables

Variable scope defines where a variable is accessible in a program, impacting data flow by controlling visibility and lifetime.

- **Types of Scope**:
  - **Local**: Variables defined inside a function, accessible only within it.
  - **Global**: Variables defined outside functions, accessible everywhere (unless shadowed).
  - **Nonlocal**: Variables in an enclosing function (used in nested functions).
  - **Built-in**: Predefined names (e.g., len, sum).
- **Rules**:
  - Python uses **LEGB** (Local, Enclosing, Global, Built-in) to resolve variable names.
  - Assignment creates a local variable unless explicitly declared global or nonlocal.
  - Reading a variable searches LEGB order; writing defaults to local unless modified.
- **Local Scope Example**:

```
def calculate():
    x = 10  # Local variable
    return x
```

Program Flow: x created in function → returned → inaccessible outside.

Data Flow: x = 10 → returned as 10.

- **Global Scope Example**:

  counter = 0 # Global variable

  def increment():

    global counter # Declare global

    counter += 1

    return counter

Program Flow: Function accesses/modifies counter → increments → returns.

Data Flow: counter (0) → incremented (1) → returned.

🧪 Example:

```
x = 10  # Global variable
def sample():
    x = 5  # Local variable
    print("Inside function:", x)
sample()
print("Outside function:", x)
```

Key Points:

- Avoid overusing global; prefer passing parameters.
- Local variables are safer, preventing unintended changes.
- Scope errors (e.g., accessing undefined variables) raise NameError.

6.6 Lambda Functions (Anonymous Functions)

Lambda functions are small, one-line anonymous functions often used with **map()**, **filter()**, and **reduce()**.

🔹 Syntax:

```
lambda arguments: expression
```

No def or return; result is the expression's value.

Can have multiple arguments, but only one expression.

🧪 Example:

```
# Lambda function to square numbers

square = lambda x: x * x

print(square(5))  # Output: 25
```

Program Flow: Defines function → callable like square(5) → returns 25.

Data Flow: x → squared → returned.

With map:

- Applies a function to each item in an iterable, returning a map object (convertible to list, tuple, etc.).

```
numbers = [1, 2, 3]

squared = list(map(lambda x: x ** 2, numbers))
```

Program Flow: map applies lambda to each element → collects results.

Data Flow: [1, 2, 3] → [1, 4, 9].

With filter:

- Filters elements where a function returns True, returning a filter object.

```
numbers = [1, 2, 3, 4]

evens = list(filter(lambda x: x % 2 == 0, numbers))
```

Program Flow: filter tests each element → keeps True results.

Data Flow: [1, 2, 3, 4] → [2, 4].

With reduce:

- From functools module; applies a function pairwise to reduce an iterable to a single value.

```
from functools import reduce

numbers = [1, 2, 3, 4]

product = reduce(lambda x, y: x * y, numbers)
```

Program Flow: reduce pairs elements (e.g., (1*2), then (2*3), then (6*4)) → returns result.

Data Flow: [1, 2, 3, 4] → 24.

6.7 Using Lambda with Functional Methods

- `map(function, iterable)`

Applies a function to every item in an iterable.

```
numbers = [1, 2, 3, 4]

squares = list(map(lambda x: x**2, numbers))

print(squares)
```

- `filter(function, iterable)`

Filters items for which the function returns True.

```
even = list(filter(lambda x: x % 2 == 0, numbers))

print(even)
```

- `reduce(function, iterable)`

Applies function cumulatively to the items (requires `functools`).

```
from functools import reduce

product = reduce(lambda x, y: x * y, numbers)

print(product)
```

6.8 DocString (Function Documentation)

DocStrings provide documentation inside functions, classes, or modules. Docstrings are string literals that document a function, module, or class, appearing as the first statement after the definition. They enhance code readability and support tools like help() or IDEs.

Syntax:

- Enclosed in triple quotes (""" or ''').
- Can be single-line or multi-line.

```
def multiply(a, b):
    """Return the product of two numbers.
    Args:
        a (int/float): First number.
        b (int/float): Second number.
    Returns:
        int/float: Product of a and b.
    """
    return a * b
```

Program Flow: Ignored by interpreter; no impact on execution.

Data Flow: No effect; purely for documentation.

Accessing:

```
doc = multiply.__doc__  # Retrieves docstring
```

6.9 Modules and Standard Modules

A **module** is a file containing Python definitions and functions. Python has many **built-in modules** like `math`, `random`, `datetime`, etc.

- Creating and Importing a Custom Module

Save code in a .py file (e.g., mymodule.py).

`my_module.py`:

```
def welcome(name):

    return f"Welcome, {name}!"
```

In main script:

```
import my_module

print(my_module.welcome("Alice"))
```

Program Flow: Imports module → accesses function → calls it.

Data Flow: Inputs → processed by module's function → returned.

- Using Standard Modules

math: Mathematical functions and constants.

```
import math

value = math.sqrt(16)  # Returns 4.0

pi = math.pi           # 3.141592653589793
```

random: Random number generation.

```
import random

num = random.randint(1, 10)  # Random int from 1 to 10
```

datetime: Date and time handling.

```
from datetime import datetime

now = datetime.now()  # Current timestamp
```

os: Operating system interactions

```
import os

cwd = os.getcwd()  # Current working directory
```

sys: System-specific parameters

```
import sys

version = sys.version  # Python version
```

🛠️ Hands-on: Functions, Arguments, Lambda, and Modules

```
# Custom function with all types of arguments
def describe_person(name, *hobbies, **details):
    """Describes a person using args and kwargs"""
    print(f"Name: {name}")
    print("Hobbies:", hobbies)
    print("Details:", details)
describe_person("Ravi", "Reading", "Cycling", age=30, city="Delhi")
# Lambda with map, filter, reduce
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x*x, numbers))
even = list(filter(lambda x: x % 2 == 0, numbers))
from functools import reduce
summation = reduce(lambda x, y: x + y, numbers)
print("Squared:", squared)
print("Even numbers:", even)
print("Sum:", summation)
# Using a standard module
import random
print("Random number between 1 and 10:", random.randint(1, 10))
```

✅ Conclusion

Functions are the foundation of any Python program. From defining reusable blocks of logic to passing complex arguments, using lambda expressions, and working with modules — mastering functions enhances your coding efficiency and scalability.

🗂️ 7. File Handling in Python

🎯 7.1 Objective:

This module helps you understand how to **interact with files in Python**. You'll learn to:

- Open, create, read, write, update, and delete files
- Use file modes effectively
- Manage file resources safely using context managers

📂 7.2 File Handling Basics

Python allows you to **perform file operations** such as **create**, **open**, **read**, **write**, **append**, and **delete** files. This is essential for tasks like saving user data, reading configurations, and writing logs. File handling involves interacting with files on the filesystem to store, retrieve, or modify data. It's essential for persistent data storage and integrates with program flow through sequential or conditional execution and data flow by moving data between memory and files. Python provides built-in functions and context managers for robust file operations.

📌 7.3 File Opening Modes

When opening a file, Python requires a **mode** that tells it what operation to perform. File opening modes specify how a file should be accessed or modified when opened. They determine the operations (read, write, etc.) and cursor behavior.

- Mode

Mode	Description	File Exists?
`'r'`	Read-only (default), file must exist	Must exist
`'w'`	Write-only, **creates** file or **overwrites** existing	Created if not exists
`'a'`	Append-only, creates if not exists	Created if not exists
`'x'`	Create a new file, fails if file exists	Create
`'b'`	Binary mode (used with other modes)	Must exist
`'+'`	Read and Write mode	Must exist

Program Flow:

- Mode determines allowed operations (e.g., r prevents writing).
- Errors (e.g., FileNotFoundError for r if file missing) alter flow if unhandled.

Data Flow:

- Mode affects data movement (e.g., r reads data into memory, w writes from memory to file).

Example:

```
# Open file in different modes

file_r = open("data.txt", "r")    # Read mode

file_w = open("data.txt", "w")    # Write mode

file_a = open("data.txt", "a")    # Append mode
```

🧰 7.4 Context Manager (`with` Statement)

The **`with` statement** is used to handle files properly. It automatically **closes the file**, even if an error occurs. A **context manager** ensures proper resource management, especially for files, by automatically handling setup and cleanup (e.g., closing files). The with statement is Python's primary context manager for file handling.

🔹 **Syntax:**

```
with open("filename.txt", "mode") as file: # perform file operations

pass # File auto-closes after block
```

open(): Creates a file object.

as file: Assigns the object to a variable.

encoding: Optional (e.g., "utf-8" for text files).

Block: Indented operations; file remains open.

Exit: File closes automatically, even if errors occur.

🔸 **Example:**

```
# Write using context manager

with open("example.txt", "w", encoding="utf-8") as file:

    file.write("Hello, Python!")
```

- Program Flow: Opens → writes → auto-closes.
- Data Flow: String "Hello, Python!" → written to example.txt.

- Benefits:
  - Prevents resource leaks (unclosed files).
  - Simplifies error handling.
  - Cleaner code compared to manual open()/close().
  - Program Flow: Opens → writes → auto-closes.
  - Data Flow: String "Hello, Python!" → written to example.txt.

🔧 7.5 File Operations in Detail

Python supports several file operations: open, create, read, write, update, and delete. Each manipulates files differently, affecting program and data flow.

7.5.1. Open a File

Description: Opens a file for reading, writing, or other operations, returning a file object.

Function: open(filename, mode, encoding=None).

Modes: See table above (r, w, a, etc.).

Program Flow: Initiates file access; must be followed by operations and closed (unless using with).

Data Flow: Prepares file for data transfer (no data moved yet).

Example

```
# Open for reading

file = open("notes.txt", "r", encoding="utf-8")

# Operations would follow

file.close()
```

7.5.2. Create a File

Description: Creates a new file or overwrites an existing one, depending on mode.

Modes:

- w, w+: Creates if not exists; overwrites if exists.
- a, a+: Creates if not exists; appends if exists.

Program Flow: Opening in w/a creates file → ready for writing/appending.

Data Flow: No data written yet; file initialized.

```
file = open("newfile.txt", "x")  # 'x' creates a new file, fails if exists
```

7.5.3. Read from a File

Description: Retrieves data from a file into memory.

Methods:

- read(): Reads entire file as a string.
- readline(): Reads one line (up to newline or EOF).
- readlines(): Reads all lines into a list of strings.
- Iterable: Loop over file object for line-by-line reading (memory-efficient).

Modes: r, r+, w+, a+ (cursor position matters).

Program Flow: Opens → reads → processes data → closes.

Data Flow: File content → memory (as string, list, or lines).

Example

```
with open("sample.txt", "r") as f:
    content = f.read()
    print(content)
```

You can also use:

- `readline()` – reads one line at a time
- `readlines()` – returns all lines as a list

7.5.4. Write to a File

Description: Writes data from memory to a file.

Methods:

- write(text): Writes a string; returns number of characters written.
- writelines(lines): Writes a list of strings (no auto-newlines).

Modes: w, a, r+, w+, a+.

- w, w+: Overwrites or creates.
- a, a+: Appends.
- r+: Writes at cursor (overwrites existing content).

Program Flow: Opens → writes → updates file → closes.

Data Flow: Memory (string, list) → file.

Example

```
with open("sample.txt", "w") as f:

    f.write("This will overwrite the file.")
```

7.5.5. Append / Update a File

Description: Modifies existing file content (no direct "update" method; achieved via read/write).

Approach:

- Read content → modify in memory → write back (often with r+ or w).
- For appending, use a mode (technically an update).

Modes: r+, w+, a, a+.

Program Flow:

- Open → read → modify data → seek (if needed) → write → close.
- Alternatively, read all → rewrite entire file.

Data Flow: File → memory → modified → file.

Example:

```
with open("sample.txt", "a") as f:

    f.write("\nAdding a new line.")
```

Example2

```
# Update specific part

with open("data.txt", "r+", encoding="utf-8") as file:

    content = file.read()          # Read all

    file.seek(0)                   # Move cursor to start

    file.write(content.replace("old", "new"))  # Overwrite with modified

    file.truncate()                # Remove leftover content
```

7.5.6. Delete a File

Description: Removes a file from the filesystem (not a file object method; uses os module).

Function: os.remove(filename) or os.unlink(filename).

Module: import os.

Program Flow: Check file existence (optional) → delete → continue.

Data Flow: No data movement; file is erased.

Example:

```
import os

if os.path.exists("sample.txt"):

    os.remove("sample.txt")

else:

    print("The file does not exist.")
```

👩‍💻 7.6 Hands-on: Complete File Operations

```
import os

# 1. Create a file and write initial content

with open("demo.txt", "w") as f:

    f.write("Welcome to Python file handling!\n")

# 2. Append additional content

with open("demo.txt", "a") as f:

    f.write("This is an appended line.\n")

# 3. Read the file content

with open("demo.txt", "r") as f:

    print("Reading File Content:\n")

    print(f.read())

# 4. Update (append) using a variable

new_line = "Appending from a variable.\n"

with open("demo.txt", "a") as f:

    f.write(new_line)

# 5. Delete the file (cleanup)

if os.path.exists("demo.txt"):

    os.remove("demo.txt")
```

```
    print("\nFile deleted successfully.")

else:

    print("\nFile does not exist.")
```

✅ 7.7Conclusion

File handling in Python is a fundamental skill for managing data persistence. Using **context managers** ensures clean, error-free file access. Python supports all standard file operations with simple syntax and powerful libraries.

File Opening Modes:

- r (read), w (write), a (append), r+ (read/write), etc.
- b for binary, t for text; affects operation and cursor.

Context Manager:

- with ensures files close automatically.
- Simplifies error handling and cleanup.

File Operations:

- **Open**: Access file with open().
- **Create**: Use w, a, w+, a+ to create/overwrite.
- **Read**: read(), readline(), readlines(), or iterate.
- **Write**: write(), writelines() to store data.
- **Update**: Read → modify → write; or append with a.
- **Delete**: os.remove() to erase files.

Program Flow:

- Sequential: open → operate → close.
- Context manager streamlines flow; errors redirect via exceptions.

Data Flow:

- Memory ↔ file (read: file to memory; write: memory to file).
- Update modifies in memory before rewriting.
- Delete removes file without data transfer.

⚠️ 8. Exception Handling in Python

Exception handling allows programs to manage errors gracefully, ensuring robust execution by catching and responding to unexpected events. Exceptions are objects that represent errors or abnormal conditions, enabling developers to control program flow and maintain data integrity.

🎯 8.1 Objective:

In this module, you will learn how to **detect and handle errors** during the execution of Python programs. You will also learn how to define and raise your own **custom exception classes** to handle user-specific errors effectively.

🧨8.2 Types of Errors in Python

Errors in Python disrupt program execution and are categorized as:

Syntax Errors:

- Caused by incorrect syntax (e.g., missing colon, unmatched parentheses).
- Detected during parsing, before execution.

Example:

```
if True  # Missing colon
   print("Error")
```

Output: SyntaxError: expected ':'

Exceptions (Runtime Errors):

- Occur during execution due to invalid operations or conditions.
- **Examples:**
    - ZeroDivisionError: Division by zero (e.g., 5 / 0).
    - TypeError: Invalid operation for types (e.g., "1" + 1).
    - ValueError: Valid type, wrong value (e.g., int("abc")).
    - FileNotFoundError: Accessing nonexistent file.
    - IndexError: Invalid list index (e.g., [1, 2][3]).
    - KeyError: Missing dictionary key (e.g., d["missing"]).
- Handled using exception handling constructs.
- Example:

result = 10 / 0 # Raises ZeroDivisionError

Logical Errors:

- Not true errors; program runs but produces incorrect results due to flawed logic.
- Example: Incorrect formula (e.g., area = side * side for a rectangle).
- Not handled by exception handling; requires debugging.

🛡️ 8.3 Exception Handling Constructs

Python uses try, except, else, and finally to manage exceptions, controlling program flow and data flow during errors.

✅ 8.3.1 `try ... except`

Purpose: Catches specific or general exceptions to prevent program crashes.

Syntax

```
try:
    # Code that might raise an exception
    statement
except ExceptionType:
    # Handle specific exception
    statement
except:
    # Handle any other exception (optional)
    Statement
```

Program Flow:

- Execute try block.
- If an exception occurs, jump to matching except block.
- Skip remaining try code; continue after except.
- If no exception, skip all except blocks.

Example

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("Invalid input; please enter a number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

Flow: Try converting input → if ValueError, print error; if ZeroDivisionError, print error; else compute result.

✅ 8.3.2 `try ... except ... finally`

The `finally` block executes **no matter what**—whether an exception occurred or not.

Purpose: Ensures cleanup code runs regardless of exceptions.

Syntax:

```
try:
   # Risky code
   statement
except ExceptionType:
   # Handle exception
```

```
    statement
finally:
    # Always execute (e.g., cleanup)
    statement
```

Program Flow:

- Execute try → if exception, run except → run finally.
- No exception: Complete try → run finally.
- finally executes even if return or unhandled exception occurs.

Data Flow:

- try processes data → except handles errors → finally ensures final state (e.g., close resources).

Example:

```
try:
    file = open("data.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found.")
finally:
    try:
        file.close()
    except NameError:
        pass  # File never opened
```

Flow: Try opening/reading → if FileNotFoundError, print error → always attempt to close file.

✅ 8.3.3 `try ... except ... else`

The `else` block runs **if no exception occurs**.

- **Purpose**: Runs code only if no exceptions occur in try.

Syntax:

```
try:
   # Risky code
   statement
except ExceptionType:
   # Handle exception
   statement
else:
   # Run if no exception
   statement
```

Program Flow:

- Execute try → if exception, run except → skip else.
- No exception: Complete try → run else → continue.

Data Flow:

- try processes data → except handles errors → else continues processing if successful.

Example:

```
try:
    num = int(input("Enter a number: "))
except ValueError:
```

```
    print("Invalid input; please enter a number.")
else:
    square = num * num
    print(f"Square: {square}")
```

Flow: Try converting input → if ValueError, print error; else compute and print square.

Data: Input → num → square (or error message).

🔁 8.4 Multiple Exceptions

Python allows catching multiple exceptions in a single except block or using multiple except blocks for specific handling.

Syntax (Single except):

```
try:
   statement
except (ExceptionType1, ExceptionType2):
   statement
```

Syntax (Multiple except):

```
try:
   statement
except ExceptionType1:
   statement
except ExceptionType2:
   statement
```

Program Flow:

- First matching except handles the exception.
- Order matters: Specific exceptions before general (e.g., ValueError before Exception).

Data Flow:

- Exception interrupts data → specific handler processes or recovers.

Example:

```
try:
    value = int(input("Enter a number: "))
    result = 100 / value
except (ValueError, TypeError):
    print("Invalid input; enter a valid number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

Flow: Try input and division → catch ValueError/TypeError together → catch ZeroDivisionError separately.

Data: Input → value → result (or error message).

🚨 8.5 Raising Exceptions

You can manually raise exceptions using the `raise` keyword. The raise keyword triggers exceptions explicitly, allowing developers to enforce conditions or signal errors.

Syntax:

```
raise ExceptionType("Error message")
```

Program Flow:

- raise interrupts execution → jumps to nearest except or crashes if unhandled.

Data Flow:

- Data processing stops → error message propagates.

Example:

```
def divide(a, b):
    if b == 0:
        raise ValueError("Denominator cannot be zero.")
    return a / b
try:
    result = divide(10, 0)
except ValueError as e:
    print(f"Error: {e}")
```

- **Flow: Call divide → if b == 0, raise ValueError → catch and print.**
- **Data: a, b → error message if b == 0.**

👩‍💻 8.6 User-Defined Exceptions

You can define your own exception class by **inheriting from `Exception`**. Custom exception classes allow tailored error handling by extending built-in Exception or its subclasses.

Syntax:

```
class CustomError(Exception):
    """Custom exception class."""
    pass
```

Program Flow:

- Define class → raise instance in code → catch with try-except.

Data Flow:

- Custom data (e.g., attributes) → carried by exception → handled.

Example:

```
class NegativeNumberError(Exception):
    """Exception for negative numbers."""
    def __init__(self, value, message="Negative numbers not allowed."):
        self.value = value
        self.message = message
        super().__init__(self.message)
def check_positive(num):
    if num < 0:
        raise NegativeNumberError(num)
    return num
```

```
try:
    result = check_positive(-5)
except NegativeNumberError as e:
    print(f"Error: {e.message} Value: {e.value}")
```

- **Flow**: Call check_positive → if negative, raise NegativeNumberError → catch and print details.
- **Data**: num → exception object with value, message → printed.

🧪 8.7 Hands-on: Exception Handling in Python

```
# Program demonstrating various exception types
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Error: Division by zero.")
    else:
        print("Result:", result)
    finally:
        print("End of division attempt.")
divide(10, 0)
# Raising and catching a custom exception
class UnderAgeError(Exception):
    pass
```

```
def validate_voter(age):
    if age < 18:
        raise UnderAgeError("You must be at least 18 to vote.")
    return "Eligible to vote"
try:
    print(validate_voter(16))
except UnderAgeError as e:
    print("Custom Exception:", e)
```

✅ 8.8 Conclusion

Exception handling ensures that your programs can **respond to unexpected conditions** without crashing. Python provides both **built-in** and **user-defined** ways to handle errors, making your code more robust and user-friendly.

🧱 9. Object-Oriented Programming (OOPs) in Python

🎯 9.1 Objective:

In this module, you will gain a comprehensive understanding of **Object-Oriented Programming (OOP)** in Python. OOP allows you to design your program around **real-world objects**, encapsulating data and behavior into a single entity. Python fully supports OOP, making it ideal for scalable and maintainable code design.

📘 9.2 What is Object-Oriented Programming?

Object-Oriented Programming (OOP) is a programming paradigm that uses **objects** and **classes** to model and structure code. It promotes code **reuse**, **modularity**, and **extensibility**. Object-oriented programming (OOP) in Python helps you structure your code by grouping related data and behaviors into objects. You start by defining classes, which act as blueprints, and then create objects from them. OOP simplifies modeling real-world concepts in your programs and enables you to build systems that are more reusable and scalable.

Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviors are bundled into individual objects.

For example, an object could represent a person with properties like a name, age, and address and behaviors such as walking, talking, breathing, and running. Or it could represent an email with properties like a recipient list, subject, and body and behaviors like adding attachments and sending.

Put another way, object-oriented programming is an approach for modeling concrete, real-world things, like cars, as well as relations between things, like companies and employees or students and teachers. OOP models real-world entities as software objects that have some data associated with them and can perform certain operations.

OOP also exists in other programming languages and is often described to center around the four pillars, or four tenants of OOP:

🧩9.3 Key OOP Concepts in Python

🔐 9.3.1 Encapsulation

Encapsulation is the practice of hiding internal object details and only exposing necessary parts. It's achieved using private or protected attributes and methods.Encapsulation allows you to bundle data (attributes) and behaviors (methods) within a class to create a cohesive unit. By defining methods to control access to attributes and its modification, encapsulation helps maintain data integrity and promotes modular, secure code.

Syntax

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # Private variable
    def deposit(self, amount):
        self.__balance += amount
    def get_balance(self):
        return self.__balance
acc = Account("Alice", 1000)
acc.deposit(500)
print(acc.get_balance())  # Output: 1500
# print(acc.__balance)    # Error: Cannot access private variable
```

🧬 9.3.2 Inheritance

Inheritance allows a class (**child**) to inherit attributes and methods from another class (**parent**), promoting code reuse.

Syntax

```
class Vehicle:
    def __init__(self, brand):
        self.brand = brand
    def start(self):
        print(f"{self.brand} is starting...")
class Car(Vehicle):  # Inherits from Vehicle
    def play_music(self):
        print("Playing music in the car.")
c = Car("Honda")
c.start()
c.play_music()
```

🔁 9.3.3 Polymorphism

Polymorphism allows the **same method name** to behave differently in different classes.Polymorphism allows you to treat objects of different types as instances of the same base type, as long as they implement a common interface or behavior. Python's duck typing make it especially suited for polymorphism, as it allows you to access attributes and methods on objects without needing to worry about their actual class.

Syntax:

```
class Vehicle:
    def __init__(self, brand):
        self.brand = brand
```

```
    def start(self):
        print(f"{self.brand} is starting...")

class Car(Vehicle):  # Inherits from Vehicle
    def play_music(self):
        print("Playing music in the car.")

c = Car("Honda")
c.start()
c.play_music()
```

Example 1: Method Overriding

```
class Animal:
    def speak(self):
        print("Animal makes a sound")
class Dog(Animal):
    def speak(self):
        print("Dog barks")
class Cat(Animal):
    def speak(self):
        print("Cat meows")
for animal in [Dog(), Cat(), Animal()]:
    animal.speak()
```

Example 2: Duck Typing

```
class Bird:
    def fly(self):
        print("Bird is flying.")
class Airplane:
    def fly(self):
        print("Airplane is flying.")
def lift_off(entity):
    entity.fly()
lift_off(Bird())
lift_off(Airplane())
```

🛠️ 9.4 Types of Methods in Python

Python supports three main types of methods inside a class:

Type	Description
Instance Method	Operates on object (`self`) and can access instance attributes
Class Method	Works with class (`cls`), used for factory patterns
Static Method	Independent utility function, no access to class or instance

Example:

```
class MyClass:
    class_var = "Shared"
    def __init__(self, value):
        self.value = value
    def instance_method(self):  # Regular method
        return f"Instance value: {self.value}"
    @classmethod
    def class_method(cls):  # Class method
        return f"Class variable: {cls.class_var}"
    @staticmethod
    def static_method():  # Static method
        return "Static method called"
obj = MyClass("Python")
print(obj.instance_method())
print(MyClass.class_method())
print(MyClass.static_method())
```

👨‍💻 9.5 Hands-on: OOP Concepts in One Example

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.__age = age  # Encapsulated
    def greet(self):
```

```
        print(f"Hi, I'm {self.name} and I'm {self.__age} years old.")
    def get_age(self):
        return self.__age
# Inheritance
class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.student_id = student_id
    def greet(self):  # Polymorphism - Overriding
        print(f"I'm student {self.name}, ID: {self.student_id}")
# Using methods
p = Person("Ravi", 40)
p.greet()
s = Student("Anita", 20, "S123")
s.greet()
# Static method
class Utility:
    @staticmethod
    def add(x, y):
        return x + y
print("Sum:", Utility.add(5, 7))
```

✅ 9.6 Conclusion

OOP in Python provides a powerful structure to model real-world entities in code. Mastering concepts like **encapsulation**, **inheritance**, and **polymorphism** enables cleaner, more maintainable, and modular code.

🎒 10. Student Management System

🧾 Project Objective:

This project allows you to **add**, **display**, **search**, and **delete** student records using **Object-Oriented Programming** principles such as **encapsulation**, **inheritance**, and **modularization**.

🧩 OOP Concepts Covered:

Feature	Concept
`Student` class	Encapsulation
`Database` class	Abstraction, Composition
`search_student`, `delete_student`	Polymorphism (different behavior with same method call)
`@staticmethod`	Utility methods
Inheritance	(Optional extension shown below)

```
# ---------------------------
# Student Management System
# ---------------------------
# Student class using encapsulation
class Student:
    def __init__(self, name, roll_no, course):
        self.__name = name
        self.__roll_no = roll_no
        self.__course = course
    def get_details(self):
        return {
            "Name": self.__name,
            "Roll Number": self.__roll_no,
            "Course": self.__course
        }
    def get_roll_no(self):
        return self.__roll_no
    def __str__(self):
        return f"Name: {self.__name}, Roll No: {self.__roll_no}, Course: {self.__course}"
# Database class handling student objects
class StudentDatabase:
    def __init__(self):
        self.students = []
    def add_student(self, student):
        self.students.append(student)
        print("✅ Student added successfully.")
```

```
    def display_students(self):
        if not self.students:
            print("⚠️ No students to display.")
        else:
            print("\n📋 All Students:")
            for student in self.students:
                print(student)
    def search_student(self, roll_no):
        for student in self.students:
            if student.get_roll_no() == roll_no:
                print("🔍 Student Found:")
                print(student)
                return
        print("❌ Student not found.")
    def delete_student(self, roll_no):
        for student in self.students:
            if student.get_roll_no() == roll_no:
                self.students.remove(student)
                print("🗑️ Student deleted.")
                return
        print("❌ Student not found.")
    @staticmethod
    def welcome_message():
        print("🎓 Welcome to Student Management System\n")
# -------- Main Program --------
def main():
    StudentDatabase.welcome_message()
```

```
db = StudentDatabase()
while True:
    print("\n1. Add Student")
    print("2. Display All Students")
    print("3. Search Student by Roll No")
    print("4. Delete Student by Roll No")
    print("5. Exit")
    choice = input("Enter your choice: ")
    if choice == '1':
        name = input("Enter name: ")
        roll_no = input("Enter roll number: ")
        course = input("Enter course: ")
        s = Student(name, roll_no, course)
        db.add_student(s)
    elif choice == '2':
        db.display_students()
    elif choice == '3':
        roll_no = input("Enter roll number to search: ")
        db.search_student(roll_no)
    elif choice == '4':
        roll_no = input("Enter roll number to delete: ")
        db.delete_student(roll_no)
    elif choice == '5':
        print("👋 Exiting program.")
        break
    else:
        print("⚠️ Invalid choice. Try again.")
```

```
# Run the program
if __name__ == "__main__":
    main()
```

🚀 Features Implemented:

- Fully object-oriented
- Realistic encapsulation of student data
- Clean and extensible design
- Use of static method for a global utility
- Polymorphic behavior in actions like search/delete

🧠 Suggested Enhancements:

- Add inheritance (e.g., `GraduateStudent` or `ScholarshipStudent`)
- Save to/load from file using file handling
- GUI with `Tkinter` or Web interface using `Flask`